Моделирование и анализ информационных систем. Т. 25, № 4 (2018), с. 347–357
Modeling and Analysis of Information Systems. Vol. 25, No 4 (2018), pp. 347–357

Параллельное программирование

DOI: 10.18255/1818-1015-2018-4-347-357

УДК 004.052.42

Оптимизация инварианта цикла в языке Пифагор

Васильев В. С., Легалов А. И.

получена 15 марта 2018

Аннотация. В работе рассматриваются методы преобразования программ, эквивалентные оптимизации инварианта цикла, применительно к функционально-потоковой модели параллельных вычислений, реализованной в языке программирования Пифагор. В императивных языках при оптимизации инварианта из цикла выносятся вычисления, не зависящие от изменяемых в нем переменных. Особенностью языка функционально-потокового параллельного программирования Пифагор является отсутствие явно задаваемых циклических вычислений (оператора цикла). Тем не менее, повторяющиеся вычисления в этом языке можно задать рекурсивно или за счет применения специфических языковых конструкций (параллельных списков). Оба механизма обеспечивают возможность параллельного выполнения. В случае оптимизации рекурсивной функции повторяющиеся операции выносятся во вспомогательную функцию, а основная функция выполняет лишь вычисление инварианта. При оптимизации внутри параллельных списков вычисление инварианта перемещается в дополнительную функцию, содержащую вызов функции, использующую данный параллельный список. В статье приводится определение «инварианта» применительно к языку Пифагор, алгоритмы его оптимизации, а также примеры программ, их графовых представлений (граф программных зависимостей) до и после оптимизации. Алгоритм оптимизации, описанный для вычислений над параллельными списками, применим только для языка Пифагор, так как опирается на специфические структуры данных и модель вычислений этого языка. Вместе с тем, алгоритм преобразования рекурсивных функций может быть применим и для других языков программирования.

Ключевые слова: функционально-потоковое параллельное программирование, язык программирования Пифагор, оптимизация кода, оптимизация циклов, оптимизация инварианта, граф программных зависимостей

Для цитирования: Васильев В. С., Легалов А. И., "Оптимизация инварианта цикла в языке Пифагор", *Моделирование и анализ информационных систем*, **25**:4 (2018), 347–357.

Об авторах:
Васильев Владимир Сергеевич, orcid.org/0000-0002-3340-6678, аспирант,
Сибирский федеральный университет, Институт космических и информационных технологий
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: vsvasilev@sfu-kras.ru

Легалов Александр Иванович, orcid.org/0000-0002-5487-0699, д-р техн. наук, профессор,
Сибирский федеральный университет, Институт космических и информационных технологий
ул. Академика Киренского, 26, г. Красноярск, 660074 Россия, e-mail: legalov@mail.ru

Благодарности:
Исследование выполняется при финансовой поддержке РФФИ в рамках научного проекта № 17-07-00288.

Введение

Оптимизация представляет собой процесс эквивалентного преобразования, в результате которого устраняются избыточные вычисления, не влияющие на ход выполнения программы, а следовательно, улучшаются требуемые характеристики. Одним из таких преобразований, широко используемых в императивном программировании, является оптимизация инварианта цикла [1]. Вычисление называется инвариантом цикла (loop-invariant computation), если для определенной группы операторов дает один и тот же результат независимо от того, сколько раз выполнится тело цикла. При оптимизации инварианта цикла (loop-invariant code motion) такие вычисления выносятся из тела цикла и размещаются перед ним, за счет чего сокращается объем производимых вычислений [2].

Особенности оптимизации инварианта цикла в императивных языках можно рассмотреть на следующем простом примере для языка программирования C++:

```
int inv (int n, int p) {
  int s = 0;
  int i = 0;




  while (i < n-2) {
    s += p*3;
    ++i;
  }
  return s;
}
```

```
int inv (int n, int p) {
  int s = 0;
  int i = 0;

  int inv_n = n-2;
  int inv_p = p*3;

  while (i < inv_n) {
    s += inv_p;
    ++i;
  }
  return s;
}
```

Фрагмент кода, размещенный в левой части, содержит цикл, в котором на каждой итерации вычисляются операции *n-2* и *p*3*, т. к. внутри цикла значения переменных n и p не изменяются, возможен их вынос из цикла. Оптимизированный вариант программы показан в правой части листинга.

Язык Пифагор разрабатывается как инструмент для написания архитектурно-независимых параллельных программ. В его основе лежит принцип управления вычислениями по готовности данных (dataflow) [3]. Программа на данном языке состоит из функций, каждая из которых однозначно отображается в граф потока данных (dataflow graph, ГПД), отражающий зависимости по данным между операторами [4]. Формируемый ГПД является ацикличным, так как язык применяет принцип единственного использования вычислительных ресурсов, по сути усиливающий принцип единственного присваивания [5]. Именно на этом графе можно проводить разнообразные оптимизационные преобразования, обеспечивающие в дальнейшем порождение более эффективного исполняемого кода. Специфика лежащей в основе языка модели вычислений позволяет формировать повторяющиеся вычисления либо с помощью рекурсии, либо за счет операций над параллельными списками.

1. Оптимизация инварианта в рекурсивной функции

Инвариантом рекурсивной функции, по аналогии с инвариантом цикла, будем считать вычисления, результат которых не будет изменяться в ходе повторного выполнения внутри рекурсивных вызовов. Такие вычисления зависят от констант и значений, неизменяемых между рекурсивными вызовами.

Представленная ниже на языке Пифагор функция описывает вычисления, аналогичные рассмотренному выше примеру на языке C++. На рис. 1 приведен ГПД, соответствующий этой функции. Передаваемый в функцию аргумент является списком, содержащим четыре значения, которые обозначены аналогично переменным из предшествующего примера.

```
rec_inv << funcdef X {
  i << X:1;
  n << X:2;
  s << X:3;
  p << X:4;
  [((i, (n,2):-):[=>, <]):?]^(
    s,
    {
      block {
        next_s << (s, (p,3):*):+;
        break << ((i,1):+, n, next_s, p)
          :rec_inv;
      }
    }
  ):. >> return;
}
```

В приведенном примере:

- неизменяемыми аргументами являются *«n»* и *«p»* (второй и четвертый элементы списка *«X»* соответственно);
- инвариантами являются вычисления *«(n, 2):–»*, *«(p, 3):*»*, а также операции внутри параллельного списка *«[=>, <]»*. Из рис. 1 видно, что эти вычисления не зависят от изменяемых аргументов рекурсивной функции (*«i»* и *«s»*).

Несмотря на то что формирование параллельного списка *«[=>, <]»* формально является инвариантом, его нельзя оптимизировать, так как значение инварианта передается в функцию в качестве дополнительного аргумента, но согласно алгебре эквивалентных преобразований параллельный список при выполнении такой операции раскроется [5].

Для выделения фрагмента кода, являющегося инвариантом в рекурсивной функции, необходимо:

- построить множество *Args* из узлов, являющихся аргументами функции;

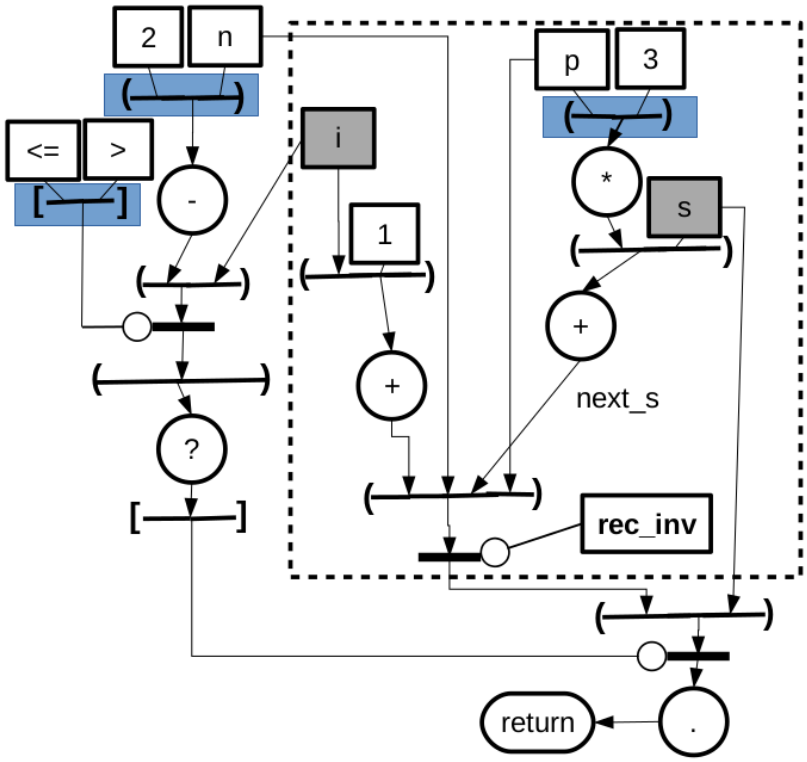

Рис. 1. ГПД рекурсивной функции с инвариантом
Fig. 1. The DFG of a recursive function with invariant

- из множества *Args* выделить подмножество *ConstArgs* неизменяемых аргументов;
- из множества операторов программы выделить инвариант (множество *Invs*), используя следующий прием: узел графа принадлежит этому множеству, если не является узлом формирования параллельного списка и имеет зависимости по данным только от констант, являющихся элементами множества *ConstArgs* и других элементов множества *Invs*.

Для оптимизации функции F, содержащей инвариант, выполняется создание вспомогательной функции G, в которую переносятся вычисления инвариантов из функции F и добавляется узел вызова функции F с передачей в нее результатов вычисления в качестве дополнительных аргументов. В функции F при этом обращения к значениям инвариантов заменяются на обращения к соответствующим аргументам. На рис. 2 приведен ГПД рассмотренной выше функции после оптимизации. В следующем листинге представлен текст функции на языке Пифагор, описывающий проведенные преобразования.

```
rec_inv_opt_h << funcdef X {
  i << X:1;
  n << X:2;
  s << X:3;
```

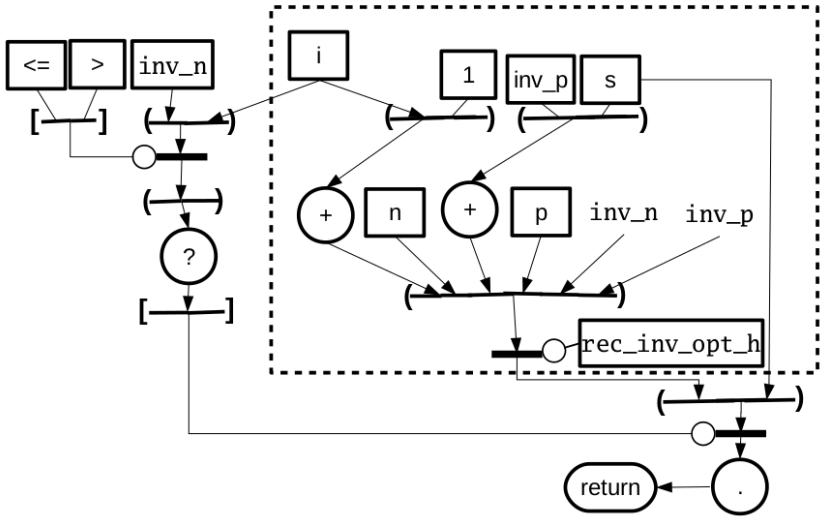

Рис. 2. ГПД рекурсивной функции после оптимизации инварианта
Fig. 2 The DFG of the recursive function after optimization of the invariant

```
  p << X:4;
  inv_n << X:5;
  inv_p << X:6;
  [((i, inv_n):[=>, <]):?]^(
    s,
    {
      block {
        next_s << (s, inv_p):+;
        break << ((i,1):+, n, next_s, p, inv_n, inv_p)
          :rec_inv_opt_h;
      }
    }
  ):. >> return;
}

rec_inv << funcdef X {
  n << X:2;
  p << X:4;
  inv_n << (n,2):-;
  inv_p << (p,3):*;
  return << (X:[], inv_n, inv_p)
    :rec_inv_optimization_h;
}
```

Следует отметить что результаты оптимизации формируются в промежуточном представлении без порождения исходных текстов функций. В данном случае и ниже

текстовое представление функций приведено только для наглядной демонстрации результатов проводимых оптимизаций.

2. Оптимизация инварианта в вычислениях с параллельными списками

В ряде случаев рекурсивные вычисления в языке Пифагор возможно заменить операциями над параллельными списками, обеспечивающими описание одновременной обработки всех данных одной функцией. При этом удается добиться более высокой эффективности распараллеливания, чем при использовании рекурсии, в частности, возможна трансляция таких конструкций в параллельные циклы ряда языков и библиотек параллельного программирования. В соответствии с алгеброй преобразований функционально-потоковой модели параллельных вычислений выполнение функции над параллельным списком преобразуется во множество параллельных операций, в которых данная функция одновременно применяется ко всем элементам этого параллельного списка:

$$[x_1, x_2, ...x_n] : f -> [x_1 : f, x_2 : f, ...x_n : f].$$

Функция выполняется над каждым элементом, который может являться как атомом, так и списком данных, независимо. Возникновение дублирующих вычислений можно рассмотреть на следующем примере. Пусть имеется список данных $XList = (x_1,\ x_2,\ ...\ x_n)$, каждый элемент которого необходимо домножить на некоторое значение Y. Для выполнения такой операции приведенным выше способом необходимо сформировать параллельный список из пар (x_i, Y) и применить к нему операцию умножения. Подобная ситуация возникает достаточно часто. При этом на месте операции умножения может стоять любая другая функция, например, *foo*. Обобщенный фрагмент кода в этом случае будет выглядеть следующим образом:

```
Len << XList:|;
YList << (Y, Len):dup;
(XList, YList):#:[]:foo;
```

В данном случае определяется длина *Len* списка *XList*, элементы которого необходимо обработать. Значение *Len* используется при выполнении операции дублирования (*dup*), формирующей список данных *YList*, состоящий из *Len* дубликатов *Y*. Списки *XList* и *YList* помещаются в список, к которому применяется операция транспонирования ($\#$), в результате чего формируется искомый список пар. Он преобразуется в параллельный список оператором *[ ]*, к которому и применяется целевая функция *foo*. Все элементы списка данных *YList* имеют одинаковое значение Y, это значение будет являться вторым аргументом функции *foo*. Таким образом, вычисления, зависящие от Y и констант, будут являться инвариантом и могут быть оптимизированы.

Для предлагаемого метода оптимизации ключевое значение имеет то, что часть данных дублируется (выполняется операция *dup*). Дублируемые данные являются неизменяемыми параметрами. Инвариантом в данном случае являются вычисления

в функции *foo*, зависящие только от констант, неизменяемых параметров и других инвариантов. Именно этот инвариант и может быть перемещен из функции *foo* в вызывающую функцию.

Для выявления неизменяемых параметров в функции, реализующей вычисления над параллельным списком, оптимизатор выполняет поиск в ГПД фрагментов, общий вид которых показан на рис. 3. При этом *YList* отражает дублированные данные, в общем случае таких списков может быть несколько.

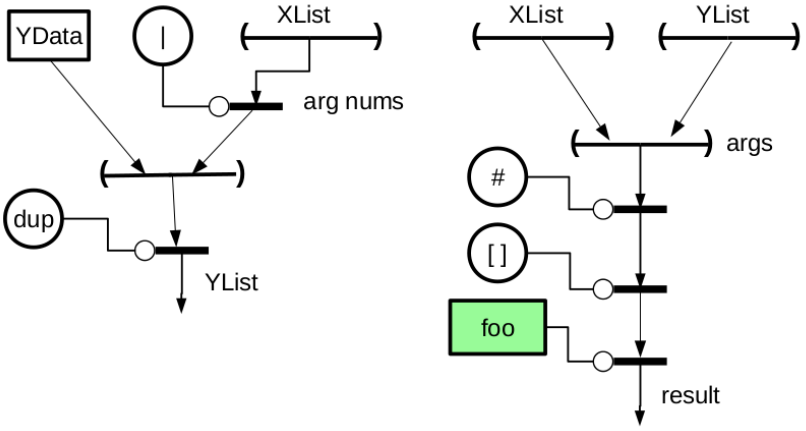

Рис. 3. Фрагмент графа, задающий повторяющиеся вычисления на списках
Fig. 3 A fragment of the graph that specifies repeating computations on lists

Поиск инвариантов внутри функции *foo* выполняется аналогично тому, как это описано для рекурсивных функций, отличается лишь способ определения неизменяемых аргументов (*ConstArgs*). Вычисление инварианта переносится в вызывающую функцию, результат вычисления передается в *foo* в качестве дополнительного аргумента.

В качестве примера рассмотрим функцию вращения трехмерной фигуры, заданной списком точек, вокруг оси. Функция принимает два аргумента — фигуру (*Figure*) и значение угла в радианах (*Alpha_rad*), на который требуется выполнить поворот. Для поворота точки используется функция *x_rotate*, для ее применения ко всем точкам определяется количество точек в фигуре (*PointsCount*), создается список из *PointsCount* углов (*Angles*). Для поворота точки матрица-строка, составленная из координат точки, умножается на матрицу вокруг соответствующей оси:

```
figure_rotate << funcdef X {
  Figure << X:1;
  Alpha_rad << X:2;

  PointsCount << Figure:|;
  Angles << (Alpha_rad, PointsCount):dup;

  return << (Figure, Angles):#:[]:x_rotate;
}
```

```
// вращение точки X в трехмерном пространстве
// вокруг оси абсцисс на Alpha_rad радиан
x_rotate << funcdef X {
  Point3d << X:1;
  Alpha_rad << X:2;

  cosA << Alpha_rad:cos;
  sinA << Alpha_rad:sin;

  rotMartix << (
    (1, 0, 0),
      (0, cosA, sinA:-),
    (0, sinA, cosA)
  );

  return << ((Point3d), rotMartix):matrix_mul;
}
```

При оптимизации функции вращения фигуры вокруг оси формируется граф, эквивалентный следующему фрагменту кода:

```
1 figure_rotate_inv_opt << funcdef X {
2   Figure << X:1;
3   Alpha_rad << X:2;
4
5   PointsCount << Figure:|;
6   Angles << (Alpha_rad, PointsCount):dup;
7
8   cosA << Alpha_rad:cos;
9   sinA << Alpha_rad:sin;
10
11  rotMartix << (
12     (1, 0, 0),
13     (0, cosA, sinA:-),
14     (0, sinA, cosA)
15   );
16
17   rotMatrices << (rotMartix, PointsCount):dup;
18
19   return << (Figure, Angles, rotMatrices):#:[]:x_rotate_inv_opt;
20 }
21
22 x_rotate_inv_opt << funcdef X {
23   Point3d << X:1;
24   Alpha_rad << X:2;
25   rotMartix << X:3;
```

```
26
27   return << ((Point3d), rotMartix):matrix_mul;
28 }
```

Инвариантом являлись вычисления синуса, косинуса угла и операции формирования матрицы вращения. В оптимизированном коде все вычисления производятся единственный раз вне зависимости от числа точек, но полученная матрица дублируется (список *rotMatrices*), так как они вынесены в функцию *figure_ rotate_ inv_ opt* (строки *5–17*).

Таким образом, если функция F содержит вызов функции *foo* над параллельным списком и имеет место инвариант, то оптимизация может выполняться по следующему алгоритму.

1. В оптимизируемой функции F необходимо найти вызов функции, на вход которой подан список, одним или несколькими из элементов которого является результат операции *dup*.

2. Сформировать список *ConstArgNums* номеров неизменяемых аргументов функции F.

3. В функции *func* найти и сохранить в списке *Invs* инварианты, с учетом того, что аргументы с номерами из списка *ConstArgNums* являются константами. Узел *Inv* принадлежит множеству *Invs* если он:

   - является актором;
   - находится в нулевом задержанном списке;
   - зависит по данным только от констант, неизменяемых аргументов (список *ConstArgNums*) или других инвариантов;
   - не является операцией группировки в параллельный список.

4. Переместить инварианты из функции *func* в функцию F.

5. Найти использования перемещенных инвариантов в функции *func*, сформировать список *UsingInvs* из узлов, значений которых не хватает в функции *func*. Для рассмотренного выше примера инвариантом будет являться, например, операция группировки в список *(0, cosA, sinA:-)*, однако напрямую это значение в функции *x_ rotate_ inv_ opt* не используется, поэтому в список *UsingInvs* он не добавляется.

6. В функцию F добавить результаты вычисления узлов множества *UsingInvs* в качестве параметров функции *func*. Для этого выполняется дублирование встроенной функцией *dup*. На примере рассматриваемой функции вращения добавляем в *figure_ rotate_ inv_ opt*:

   ```
   rotMatrices << (rotMartix, PointsCount):dup;
   ```

7. В функцию *func* добавить код получения вычисленных значений инвариантов из списка-аргумента функции. В частности для функции вращения добавим в *x_ rotate_ inv_ opt*:

```
rotMartix << X:3;
```

3. Заключение

Показана возможность проведения преобразований, эквивалентных оптимизации инварианта цикла, в программах языка Пифагор для двух случаев: рекурсивной функции и параллельных списков. Оптимизация инварианта в рекурсивной функции может быть применима к другим языкам программирования, в то время как оптимизация для параллельных списков основывается на специфических языковых конструкциях и может использоваться только в языке Пифагор. В обоих вариантах оптимизации рассматривается функция, тело которой будет выполнено многократно. В этой функции выполняется поиск фрагментов, результат вычисления которых будет одинаковым для всех обращений к функции. Эти фрагменты перемещаются из функции и вычисляются единожды (и передаются в функцию в виде дополнительных аргументов) — следовательно, преобразование сокращает объем вычислений. Представленные методы оптимизации проводятся после трансляции функций, обеспечивая тем самым архитектурно-независимый анализ разработанного кода. Все последующие трансформации функционально-потоковых параллельных программ, включая верификацию, тестирование, отладку, а также преобразование в архитектурно-зависимые представления могут проводиться после предложенных методов оптимизации.

Список литературы / References

[1] Ахо А. В., Лам М. С., Сети Р., Ульман Дж. Д., *Компиляторы: принципы, технологии и инструментарий*, 2-е изд., Вильямс, М., 2008, 1184 с.; In English: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd Edition, Addison Wesley, 2006, 1000 pp.

[2] Дортман П. А., "Подходы к оптимизации программ в системе SFP", *Программные средства и математические основы информатики*, Новосибирск, 2004, 43–49; [Dortman P. A., "Program optimizacion in SFP", *Software tools and mathematical foundations of informatics*, Novosibirsk, 2004, 43–49, (in Russian).]

[3] Легалов А. И., Матковский И. В., Кропачева М. С., Удалова Ю. В., Васильев В. М., "Технологические аспекты создания, преобразования и выполнения функционально-потоковых параллельных программ", *Научный сервис в сети Интернет: все грани параллелизма: Труды Международной суперкомпьютерной конференции*, Изд-во МГУ, М., 2013, 443–447; [Legalov A. I., Matkovsky I. V., Kropacheva M. S., Udalova Y. V., Vasiliev V. M., "Tekhnologicheskie aspekty sozdaniya, preobrazovaniya i vypolneniya funkcionalno-potokovyh parallelnyh programm", *Nauchnyy servis v seti Internet: vse grani parallelizma: Trudy Mezhdunarodnoy superkompyuternoy konferentsii*, Izd-vo MGU, M., 2013, 443–447, (in Russian).]

[4] Легалов А. И., Васильев В. С., Матковский И. В., Ушакова М. С., "Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ", *Труды Института системного программирования РАН*, **29**:5 (2017), 165–184; [Legalov A. I., Vasilyev V. S., Matkovskii I. V., Ushakova M. S., "Support tools for

creation and transformation of functional-dataflow parallel programs", *Proceedings of ISP RAS*, **29**:5 (2017), 165–184, (in Russian).]

[5] Легалов А. И., "Функциональный язык для создания архитектурно-независимых параллельных программ", *Вычислительные технологии*, **10**:1 (2005), 71–89; [Legalov A. I., "Functional language for creating of architectural independent parallel programs", *Computational Technologies*, **10**:1 (2005), 71–89, (in Russian).]

Vasilev V. S., Legalov A. I., "Loop-invariant Optimization in the Pifagor Language", *Modeling and Analysis of Information Systems*, **25**:4 (2018), 347–357.

DOI: 10.18255/1818-1015-2018-4-347-357

Abstract. The paper considers methods of program transformation equivalent to optimizing the cycle invariant, applied to the functional data-flow model implemented in the Pifagor programming language. Optimization of the cycle invariant in imperative programming languages is reduced to a displacement from the cycle of computations that do not depend on variables that are changes in the loop. A feature of the functional data flow parallel programming language Pifagor is the absence of explicitly specified cyclic computations (the loop operator). However, recurring calculations in this language can be specified recursively or by applying specific language constructs (parallel lists). Both mechanisms provide the possibility of parallel execution. In the case of optimizing a recursive function, repeated calculations are carried out into an auxiliary function, the main function performing only the calculation of the invariant. When optimizing the invariant in computations over parallel lists, the calculation of the invariant moves from the function that executes over the list items to the function containing the call. The paper provides a definition of "invariant" applied to the Pifagor language, algorithms for its optimization, and examples of program source codes, their graph representations (the program dependence graph) before and after optimization. The algorithm shown for computations over parallel lists is applicable only to the Pifagor language, because it rests upon specific data structures and the computational model of this language. However, the algorithm for transforming recursive functions may be applied to other programming languages.

Keywords: data driven functional parallel programming, Pifagor programming language, code optimization, loop optimization, invariant optimization, program dependence graph

On the authors:
Vladimir S. Vasilev, orcid.org/0000-0002-3340-6678, graduate student,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo str., Krasnoyarsk 660074, Russia, e-mail: ksv@akadem.ru

Alexander I. Legalov, orcid.org/0000-0002-5487-0699, doctor of science,
Siberian Federal University, Institute of Space and Information Technology,
26 Kirenskogo str., Krasnoyarsk 660074, Russia, e-mail: legalov@mail.ru

Acknowledgments:
The research is supported by RFBR (research project No 17-07-00288).